Recursion

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Recursion is one of the most powerful ideas in Computer Science and in Mathematics. A proper understanding of it is essential. All the lectures from here on out will really be about recursion.

- Understand how recursion is related to induction
- Know four patterns of recursion:
  Iterating, Mapping, Folding, Accumulating
- Know how to determine the time complexity of a recursive operation
- Know how to turn a linear recursion into an exponential recursion

A proof by induction works by making two steps do the work of an infinite number of steps. It's really a way of being very lazy!

- Pick a property $P(n)$ which you'd like to prove for all n.
- **Base case:** Prove $P(n)$, for $n = 1$, or whatever n's smallest value should be.
- **Induction Case:** You want to prove $P(n)$, for some general n. To do that, *assume* that $P(n-1)$ is true, and use that information to prove that $P(n)$ has to be true.

The idea is that there are an infinite number of n such that $P(n)$ is true. But with this technique you only had to prove two cases.

To Prove: Let $P(n)$ = "The sum of the first n odd numbers is n^2."

Base Case: Let $n = 1$. Then n^2 = 1, and the sum of the list $\{1\}$ is 1; therefore the base case holds.

Induction Case: Suppose you need to show that this property is true for some n. First, pretend that somebody else already did all the work of proving that $P(n-1)$ is true. Now use that to show that $P(n)$ is true, and take all the credit.

If $\{1, 3, 5, \ldots, 2n-3\} = (n-1)^2$, then add $2n-1$ ...

$$\{1, 3, 5, \ldots, 2n-3, 2n-1\} = (n-1)^2 + 2n - 1$$
$$\Rightarrow n^2 - 2n + 1 + 2n - 1 \Rightarrow n^2$$

Recursion

A recursive routine has a similar structure. You have a base case, a recursive case, and a conditional to check which case is appropriate.

- Pick a function $f(n)$ which you'd like to compute for all n.
- **Base case:** Compute $f(n)$, for $n = 1$, or whatever n's smallest value should be.
- **Recursive Case:** Assume that someone else already computed $f(n-1)$ for you. Use that information to compute $f(n)$, and then take all the credit.

Iterating Recursion Example

Suppose you want a recursive routine that computes the nth square.

```
1 # let rec nthsq n =   (* note the rec keyword! *)
2     match n with
3     | 0 -> 0
4     | n -> (2*n-1) + (nthsq (n-1));;
5 val nthsq : int -> int = <fun>
6 # nthsq 3;;
7 - : int = 9
```

- The patterns are the conditional to check which case is active.
- Line 3 is the base case — it stops the recursion.
- Line 4 is the recursive case.

Important things about recursion

```
1 # let rec nthsq n = match n with
2   | 0 -> 0
3   | n -> (2*n-1) + (nthsq (n-1));;
```

- Your base case has to stop the computation.
- Your recursive case has to call the function with a *smaller* argument than the original call.
- Your `if` statement has to be able to tell when the base case is reached.
- Failure to do any of the above will cause an infinite loop.

Watching recursion work. . .

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3
```

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ (2*3-1) + (nthsq (3-1))
```


Recursion
Watching recursion work...
January 24, 2007
§1 **Recursion and Induction**

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
```

Illinois Institute of Technology Mattox Beckman
Recursion – p. 8

Recursion
Watching recursion work...
January 24, 2007
§1 **Recursion and Induction**

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
   nthsq 2
```

Illinois Institute of Technology Mattox Beckman
Recursion – p. 8

Recursion
Watching recursion work...
January 24, 2007
§1 **Recursion and Induction**

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
   nthsq 2 ⇒ (2*2-1) + (nthsq (2-1))
```

Illinois Institute of Technology Mattox Beckman
Recursion – p. 8

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
```



```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
   nthsq 2 ⇒ 3 + (nthsq 1)
   nthsq 1 ⇒ (2*1-1) + (nthsq (1-1))
```



```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
   nthsq 2 ⇒ 3 + (nthsq 1)
   nthsq 1 ⇒ 1 + (nthsq 0)
```



```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
   nthsq 2 ⇒ 3 + (nthsq 1)
   nthsq 1 ⇒ 1 + (nthsq 0)
   nthsq 0 ⇒ 0
```

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
   nthsq 2 ⇒ 3 + (nthsq 1)
   nthsq 1 ⇒ 1 + 0
```


Recursion
Watching recursion work...
January 24, 2007
§1 **Recursion and Induction**

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + (nthsq 2)
   nthsq 2 ⇒ 3 + 1
```

Illinois Institute of Technology Mattox Beckman
Recursion – p. 8

Recursion
Watching recursion work...
January 24, 2007
§1 **Recursion and Induction**

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 5 + 4
```

Illinois Institute of Technology Mattox Beckman
Recursion – p. 8

Recursion
Watching recursion work...
January 24, 2007
§1 **Recursion and Induction**

```
let rec nthsq n =
 match n with
 | 0 -> 0
 | n -> (2*n-1) + (nthsq (n-1))
```

```
nthsq 3 ⇒ 9
```

Illinois Institute of Technology Mattox Beckman
Recursion – p. 8

Lists

Because lists are recursive, functions that deal with lists tend to be recursive.

```
# let rec length lst = match lst with
  | [] -> 0
  | x::xs -> 1 + length xs;;
val length : 'a list -> int = <fun>
# length [2;3;4;6];;
- : int = 4
```

- The base case `[]` stops the computation.
- Your recursive case calls itself with a *smaller* argument (`xs`) than the original call.

Mapping Recursion

One common form maps a function to each of the elements.

```
# let rec doubleList lst = match lst with
  | [] -> []
  | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [4;6;8];;
- : int list = [8; 12; 16]
```


Watching the easy way

- Don't trace the whole thing. Just trace out the last step.

```
let rec length lst = match lst with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
length [2;4;6;8]
⇒ 1 + length [4;6;8]
⇒ 1 + 3
```


Trace doubleList

```
let rec doubleList lst = match lst with
  | [] -> []
  | x::xs -> 2 * x :: doubleList xs
```

```
doubleList [2;4;6;8]
⇒ 2*2 :: doubleList [4;6;8]
⇒ 2*2 :: [8;12;16]
```

Another common form "folds" a list via some function.

```
# let rec multList lst = match lst with
  | [] -> 1
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

This computes $(2 * (4 * (6 * 1)))$.

Remember the big-O notation from other CS courses.

- The question: if we have an input of size n, how long will it take to generate the output?
- Express the output time in terms of the input size, omit constants, and take the largest power.
- Common big-O times:
  - Constant time $\mathcal{O}(1)$ — input size doesn't matter
  - Linear time $\mathcal{O}(n)$ — double input $\Rightarrow$ double output
  - Quadratic time $\mathcal{O}(n^2)$ — double input $\Rightarrow$ quadruple output
  - Exponential time $\mathcal{O}(2^n)$ — increment input $\Rightarrow$ double output

- Expect most list operations to take linear time ($\mathcal{O}(n)$).
- Each step of the recursion can be done in constant time.
- Each step makes exactly one recursive call.
- List example: `multList`, `append`
- Integer example: `factorial`

- Each step of the recursion takes time proportional to the input.
- Each step of the recursion makes exactly one recursive call.
- List example: badly written "reverse".
- Integer example: Twelve days of Christmas.

```
# let rec badReverse lst = match lst with
  | [] -> []
  | x::xs -> (badReverse xs) @ [x];;
val badReverse : 'a list -> 'a list = <fun>
```

Exponential Time

- Hideous running times.
- Each step of the recursion takes constant time.
- But each recursion makes *two* recursive calls.
- Worst part: it is very simple to make a linear function into an exponential one!
- Examples: näive Fibonacci sequence (1,1,2,3,5,8,13,21,34...)

```
1 # let rec badFib n = match n with
2   | 0 -> 0
3   | 1 -> 1
4   | _ -> badFib (n-1) + badFib (n-2);;
5 val badFib : int -> int = <fun>
```


Tail Calls

Tail Position A subexpression s of expressions e, if it is evaluated, will be taken as the value of e.

- `if (x>3) then x + 2 else x - 4`
- `let x = 5 in x + 4`
- `f (x * 3)` — no tail position here.

Tail Call A function call that occurs in tail position.

- `if (h x) then h x else (x + g x)`

A non-tail-call

```
1  let start z = f z
2  let f a1 =
3    let a2 = a1 + 1 in
4      2 + g a2
5  let g a3 =
6    let a4 = a3 + 2 in
7      4 + h a4
8  let h a5 =
9    let a6 = a5 + 12 in
10     6 + a6
```

Call `start 5`. What happens?

A non-tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = a1 + 1 in
4      2 + g a2
5  let g a3 =
6    let a4 = a3 + 2 in
7      4 + h a4
8  let h a5 =
9    let a6 = a5 + 12 in
10     6 + a6
```

A non-tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = 6 in
4      2 + g 6
5  let g a3 =
6    let a4 = a3 + 2 in
7      4 + h a4
8  let h a5 =
9    let a6 = a5 + 12 in
10     6 + a6
```


A non-tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = 6 in
4      2 + g 6
5  let g a3 =
6    let a4 = 8 in
7      4 + h 8
8  let h a5 =
9    let a6 = a5 + 12 in
10     6 + a6
```


A non-tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = 6 in
4      2 + g 6
5  let g a3 =
6    let a4 = 8 in
7      4 + h 8
8  let h a5 =
9    let a6 = 20 in
10     6 + 20
```


A non-tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = 6 in
4      2 + g 6
5  let g a3 =
6    let a4 = 8 in
7      4 + h 8
8  let h a5 =
9    let a6 = 20 in
10     26
```

A non-tail-call

```
1 let start z = f 5
2 let f a1 =
3   let a2 = 6 in
4     2 + g 6
5 let g a3 =
6   let a4 = 8 in
7     4 + h 8
8 let h a5 = 26
```

What does the stack look like now?
At this point, we start returning.

A non-tail-call

```
1 let start z = f 5
2 let f a1 =
3   let a2 = 6 in
4     2 + g 6
5 let g a3 =
6   let a4 = 8 in
7     4 + 26
8 let h a5 = ...
```


A non-tail-call

```
1 let start z = f 5
2 let f a1 =
3   let a2 = 6 in
4     2 + g 6
5 let g a3 =
6   let a4 = 8 in
7     30
8 let h a5 = ...
```


A non-tail-call

```
1 let start z = f 5
2 let f a1 =
3   let a2 = 6 in
4     2 + 30
5 let g a3 = ...
6 let h a5 = ...
```

A non-tail-call

```
1 let start z = f 5
2 let f a1 =
3   let a2 = 6 in
4     32
5 let g a3 = ...
6 let h a5 = ...
```


A non-tail-call

```
1 let start z = 32
2 let f a1 = ...
3 let g a3 = ...
4 let h a5 = ...
```

We're done. But what if things were a little different?

A tail-call

```
1  let start z = f z
2  let f a1 =
3    let a2 = a1 + 1 in
4      g a2
5  let g a3 =
6    let a4 = a3 + 2 in
7      h a4
8  let h a5 =
9    let a6 = a5 + 12 in
10     a5
```

Call `start 5`. What happens?

A tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = a1 + 1 in
4      g a2
5  let g a3 =
6    let a4 = a3 + 2 in
7      h a4
8  let h a5 =
9    let a6 = a5 + 12 in
10     a5
```

A tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = 6 in
4      g 6
5  let g a3 =
6    let a4 = a3 + 2 in
7      h a4
8  let h a5 =
9    let a6 = a5 + 12 in
10     a5
```

We propagate our value to the end as before.

A tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = 6 in
4      g 6
5  let g a3 =
6    let a4 = 8 in
7      h 8
8  let h a5 =
9    let a6 = a5 + 12 in
10     a5
```

Each step takes us closer to the “base case”.

A tail-call

```
1  let start z = f 5
2  let f a1 =
3    let a2 = 6 in
4      g 6
5  let g a3 =
6    let a4 = 8 in
7      h 8
8  let h a5 =
9    let a6 = 20 in
10     20
```

Once we hit the bottom, the result propagates back.

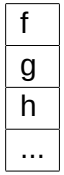
A tail-call

```
1  let start z = 20
2  let f a1 =
3    let a2 = 6 in
4      20
5  let g a3 =
6    let a4 = 8 in
7      20
8  let h a5 =
9    let a6 = 20 in
10     20
```

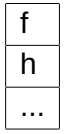
Note that it is never touched! What happens on the stack?

An important optimization

Normal call:



g tail calls h:



- When you make a function call, you have to save the return address on the stack, so we know where to return.
- Suppose f calls g, and then g calls h. What if the call to h was the very last thing g did?
- Such a call is called a *tail call*. We don't need to save the stack frame of the function making the tail call in such a case.
- This optimization can allow recursive programs to run with the same efficiency as imperative programs.

Forward Recursion

- In recursion, you split the input into the "first piece" and the "rest of the input".
- In forward recursion: the recursive call computes the result for the rest of the input, and then the function combines the result with the first piece.
- In other words, you wait until the recursive call is done to generate your result.

```
# let rec badReverse lst = match lst with
  | [] -> []
  | x::xs -> (badReverse xs) @ [x];;
val badReverse : 'a list -> 'a list = <fun>
```


Accumulating Recursion

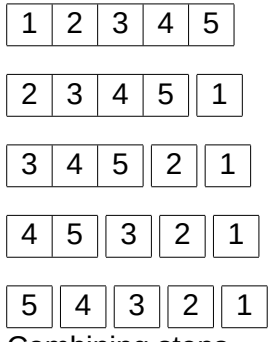
- In accumulating recursion: generate an intermediate result *now*, and give that to the recursive call.
- Usually this requires an auxiliary function.

```
# let rec goodRevAux lst acc = match lst with
  | [] -> acc
  | x::xs -> goodRevAux xs (x::acc);;
   (* notice that this one is tail recursive! *)
val goodRevAux : 'a list -> 'a list -> 'a list = <fun
# let goodReverse lst = goodRevAux lst [];;
val foo : 'a list -> 'a list = <fun>
```

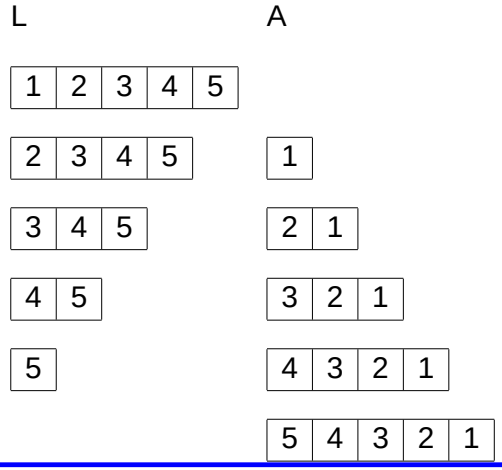
What is the running time?

Comparison

Bad Reverse



Good Reverse



1. Write an OCaml function that returns the maximum element of a list. Use forward recursion. (Assume the list always has at least 1 element.)
2. Now write the same function, but use tail recursion.
3. What is the running time of the following function?

```
# let rec doubleSum lst = match lst with
  | [] -> 0
  | x::xs -> doubleSum xs + doubleSum xs;;
val doubleSum : 'a list -> int = <fun>
```

4. Fix the above function to make it run in linear time.

Write a function that returns the maximum element of a list. Use forward recursion. (Assume the list always has at least 1 element, and assume you have a `max` function.)

```
# let rec maxlist lst = match lst with
  | [x] -> x
  | x::xs -> max x (maxlist xs);;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val maxlist : 'a list -> 'a = <fun>
```


Now write the same function, but use tail recursion.

```
# let rec maxlistaux lst a = match lst with
  | [] -> a
  | x::xs -> maxlistaux xs (max x a);;
val maxlistaux : 'a list -> 'a -> 'a = <fun>
# let maxlist (x::xs) = maxlistaux xs x;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val maxlist : 'a list -> 'a = <fun>
```


What is the running time of the following function?

```
# let rec doubleSum lst = match lst with
  | [] -> 0
  | x::xs -> doubleSum xs + doubleSum xs;;
val doubleSum : 'a list -> int = <fun>
```

It is $\mathcal{O}(2^n)$

Problem 4

Fix the above function to make it run in linear time.

```
# let rec doubleSum lst = match lst with
  | [] -> 0
  | x::xs -> 2 * doubleSum xs;;
val doubleSum : 'a list -> int = <fun>
```


Further Reading

- Forward recursion can be made to traverse a list at return time rather than call time, forming a pattern called "There and Back Again," which can do some interesting things. . . .
- Example: write a function `convolve : int list -> int list -> int list` which takes two lists $[x_1; x_2; \cdots; x_n]$ and $[y_1; y_2; \cdots; y_n]$ and produces an output list $[x_1 y_n; x_2 y_{n-2}; \cdots x_n y_1]$ where n is unknown. Use only n recursive calls, and no temporary lists.
- For the solution, see Olivier Danvy's paper There and Back Again.